فن التعامل مع الـ Structures في السي

تعلم قواعد اللعبة ثم العب افضل من الباقين .

البرت اينشتين

Mohamed Hussein

للتواصل:

للاستفسار او في حالة وجود اخطاء يرجي التواصل علي الايميل

m.hussein1389@gmail.com

Mohamed Hussein

1. لماذا ؟

- احيانا كثيرا نعمل علي برامج تستخدم الذاكرة بشكل كبير ويكون لدينا قيود(constrains) علي التعامل مع تلك الذاكرة مثل التعامل مع الانظمة المدمجة (Embedded Systems) او مثلا كتابة نواه نظام تشغيل-operating) (system kernels في حالة مطوري انظمة التشغيل(os developers) وبشكل عام هي حالات التي تكون فيها الذاكرة التي نتعامل معها محدودة لذا علينا تحسين الكود من اجل الاستخدام الامثل للذاكرة او ما يسمي

(Memory Optimization) سنري انه من خلال اعادة ترتيب عناصر الـ Structureسيقوم بتقليل حجم الذاكرة المستخدمة بنسبة تصل الي 25% مما يسمح لنا باستخدام الذاكرة المتاحة لنا بشكل افضل

2. متطلبات المحاذاة (Alignment requirements):

- اول شئ علينا فهمه هو انه في المعاجات الحديثة الطريقة التي يقوم بها مترجم السي (c compiler) بحجز اماكن في الذاكرة للبيانات عليها بعض القيود وذلك لجعل عملية القرءاة والكتابة علي الذاكرة اسرع

فاماكن البيانات في الذاكرة علي معاجات X86 ,ARM لا تكون في اي مكان في الذاكرة ما عدا المتغير من نوع char

الذي يمكن ان يكون في اي مكان (اقصد العنوان الذي يوجد به) وذلك لانه يحتاج بايت واحد فقط ,اما في حالة

متغير من نوع shortالذي يحتاج الي 2 بايت لابد ان يبدا في عنوان زوجي (even address) ,وفي حالة متغير من نوع int الذي يحتاج الي 4 بايت لابد ان يبدا في عنوان يقبل القسمة علي 4 وبالمثل في حالة المتغير من long

او double لابد ان يبدا في عنوان يقبل القسمة علي 8 وهذا ينطبق علي حالتي اذا كان المتغير باشارة او لا

(Signed or unsigned)

- ذلك لان البيانات في السي علي تلك المعالجات هي مؤشرات ذاتية المحاذاة (self-aligned Pointers) سواء علي انظمة 32 بت او 64 بت وهذا يجعل الكتابة او القرءاة من الذاكرة او مايسمي (memory access) تتم بشكل اسرع

لانها تعطي تعليمة اسمبلي واحدة فقط لجلب المتغير من الذاكرة

- بعض المعاجات القديمة تجبر برنامج السي علي عدم الالتزام بقواعد المحاذاة هذه مما يؤدي الي جعل البرنامج يعمل بشكل ابطأ واخطاء في تنفيذ الكود مثال علي ذلك المعالج Sun SPARC

3. الحشو (Padding):

- دعونا نبدا بهذ الكود الذي يقوم بحجز اماكن لعدة متغيرات في الذاكرة لنري ما يحدث

```
char *p;
char ch;
int number;
```

- اذا افترضنا اننا لا نعرف شي عن محاذاة البيانات (data alignment) فان تلك المتغيرا ت الثلاثة ستشغل وحدات تخزينية متصلة في الذاكرة مثلا في حالة انظمة 32 بت

p	ch	number
4 bytes	1 byte	4 bytes

وفي حال انظمة 64 بت

p	ch	number
8 bytes	1 byte	4 bytes

- ولكن ما يحدث فعليا هو الاتي

```
char *p; // 4 or 8 bytes
char ch; //1 byte
char pad[3]; //3 bytes
int number; //4 bytes
```

p	ch	padding	number
4 or 8 bytes	1 byte	3 byte	4 bytes

-ماذا حدث فعليا

- p يبدا في عنوان يقبل القسمة علي 4 في حال انظمة 32 بت او 8 في 64 بت (self-aligned)

- ch يبدا في العنوان التالي حيث انه يحتاج بايت واحد ويمكن تخزينه في اي عنوان

- طبقا لمتطلبات المحاذاة (Alignment requirements) فان number تحتاج ان تبدا في عنوان يقبل القسمة علي 4 لذا وضع pad بـ 3 بايت بينهما

*محتويات الحشو padding غير معروفة فليس من الضروري ان تكون اصفار

- بالمقارنة في حالة اذا كان number من نوع short

```
char *p;
char ch;
short number;
```

ستاخذ الشكل التالي في الذاكرة

```
char *p; // 4 or 8 bytes
char ch; //1 byte
char pad; //1 byte
short number; //2 bytes
```

- بالمثل اذا كان number من نوع long علي نظام 64 بت

```
char *p;
char ch;
long number;
```

-ستصبح

```
char *p; // 4 or 8 bytes
char ch; //1 byte
char pad[7]; //7 bytes
long number; //8 bytes
```

-السؤال الان كيف يمكن لنا التاكد من ذلك ؟

-الاجابة اننا قمنا بكتابة البرنامج التالي الذي يقوم بتعريف 3 متغيرات وطباعة بعض المعلومات المتعلقة بها باستخدام متغير من نوع char وجعله يشير الي عنوان من نوع int وقمنا بعد ذلك بعرض محتويات الذاكرة الخاصة بهذه المتغيرات

-لفهم نتائج البرنامج التالي هناك نقطتين هامتين لابد من معرفتهم

1-ان عنوان اي متغير يشير الي البايت الاول او (least byte) او ما يسمي بـ (little indian) بمعني اذا قمنا بانشاء متغير من نوع short واعطيناه القيمة (ox1234) وقمنا بانشاء مؤشر من نوع char واعطيناه عنوان المتغير الذي انشاناه مسبقا من نوع short فانه فعليا يشير الي القيمة (ox34) او (least byte) ولجلب القيمة الاخري علينا بزيادة هذا العنوان بمقدار واحد وذلك في حالة معاجات intel والنوع الاخر هوbig indian علي معالجات SPARC

2- ان حجم اي مؤشر لا يعتمد علي نوعه وانما علي النظام اما 4 بايت في 32 بت او 8 بايت في 64 بت , نوع المؤشر مهم فقط في العمليات الحسابية علي المتغيرات مثلا اذا كان لدينا مؤشر من نوع int وقمنا بجمع واحد عليه فانه فعليا يجمع 4لانه يعرف ان المتغير من نوع intيشغل 4 بايت في الذاكرة الجدول التالي يوضح الامر اكثر

التعريف	الكود	ما يحدث فعليا
Char *p	P++;	**P=p+1;**
Int *p	P++;	**P=p+4;**
Short *p	P++;	**P=p+2;**

وهكذا, البرنامج كالتالي

```
 /*
 * padding.c
 *
 * Created: 9/20/2016 2:02:52 PM
 *  Author: mohamed hussein
 */

#include <stdio.h>
#include <stdlib.h>

int main()
{
    unsigned char *p;
    char ch='a';
    int num=0x1234;

    p=&num;
    short i=0;

    printf("understanding padding code!\n");
    puts("addresses and sizes of variables in memory");
    printf("char *p  size is  [ %d ]  address is  [ %p ]\n",sizeof(p),&p);
    printf("char ch  size is  [ %d ]  address is  [ %p ]\n",sizeof(ch),&ch);
    printf("int num  size is  [ %d ]  address is  [ %p ]\n",sizeof(num),&num);
    puts("------------------------------------------------------------");
    puts("the content of memory containing the variables");

   for( i=0; i<15; i++)
    {
        printf("address [ %x ] contain  [ %x ] as hexadecimal and [ %c ] as
ASCII\n",p+i,*(p+i),*(p+i));
    }


    return 0;
}
```

*الكود مرفق مع الكتاب

- نتيجة البرنامج هي

```
understanding padding code!
addresses and sizes of variables in memory
char *p  size is  [ 4 ]  address is  [ 0028FF18 ]
char ch  size is  [ 1 ]  address is  [ 0028FF17 ]
int num  size is  [ 4 ]  address is  [ 0028FF10 ]
------------------------------------------------------------
the content of memory containing the variables
address [ 28ff10 ] contain  [ 34 ] as hexadecimal and [ 4 ] as ASCII
address [ 28ff11 ] contain  [ 12 ] as hexadecimal and [ ↕ ] as ASCII
address [ 28ff12 ] contain  [ 0 ] as hexadecimal and [   ] as ASCII
address [ 28ff13 ] contain  [ 0 ] as hexadecimal and [   ] as ASCII
address [ 28ff14 ] contain  [ 28 ] as hexadecimal and [ ( ] as ASCII
address [ 28ff15 ] contain  [ 4b ] as hexadecimal and [ K ] as ASCII
address [ 28ff16 ] contain  [ 48 ] as hexadecimal and [ H ] as ASCII
address [ 28ff17 ] contain  [ 61 ] as hexadecimal and [ a ] as ASCII
address [ 28ff18 ] contain  [ 10 ] as hexadecimal and [ ► ] as ASCII
address [ 28ff19 ] contain  [ ff ] as hexadecimal and [   ] as ASCII
address [ 28ff1a ] contain  [ 28 ] as hexadecimal and [ ( ] as ASCII
address [ 28ff1b ] contain  [ 0 ] as hexadecimal and [   ] as ASCII
address [ 28ff1c ] contain  [ 2 ] as hexadecimal and [ ☻ ] as ASCII
address [ 28ff1d ] contain  [ 0 ] as hexadecimal and [   ] as ASCII
address [ 28ff1e ] contain  [ e ] as hexadecimal and [ ♫ ] as ASCII
```

- سادع فهم البرنامج والنتائج لكم ولكن الشكل التالي قد يساعدكم قليلا

ملاحظات	المحتويات	العنوان
المحتويات تمثل عنوان المتغير num	0x00	**0x0028F1B**
	0x28	**0x0028FF1A**
	0xFF	**0x0028FF19**
	0x10	**0x0028FF18**
محتويات المتغير ch	‘a’	**0x0028FF17**
حشو	padding	**0x0028FF16**
حشو	padding	**0x0028FF15**
حشو	padding	**0x0028FF14**
المحتويات تمثل المتغير num	0x00	**0x0028FF13**
	0x00	**0x0028FF12**
	0x12	**0x0028FF11**
	0x34	**0x0028FF10**

- والان لجعل تلك المتغيرات تشغل مساحة اقل بدون حشو سنتقوم باعادة ترتيب تلك المتغيرات كالتالي

قبل الترتيب	بعد الترتيب
`char *p;` `char ch;` `int number;`	`char *p;` `int number;` `char ch;`
الحشو بمقدار 3 بايت	لا يوجد حشو

-اي اننا قمنا بتقليص المساحة التخزينية بدلا من 12 بايت الي 9 بايت اي بقدار 25 %

-ومع ذلك قد لا يبدو الامر مهم جدا في التعامل مع المتغيرا ت الاساسية في السي ولكن عند التعامل مع الهياكلstructures يصبح امر مهم جدا

4- الحشو في الــstrucures

-بشكل عام فان الـ struct يتم عمل محاذاة لها طبقا لاكبر عنصر موجود بها وكذلك في السي عنوان الـ structهو عنوان اول عنصر بها ولذلك لا يوجد حشو في مقدمتها (there is no leading padding)

-لنري المثال التالي

```
struct info{
     char *p;
     char ch;
     int num;
 };
```

هنا اي متغير من نوع struct info سيحدث له محاذاة علي 4 بايت في حال انظمة 32 بت علي اعتبار ان اكبر عنصر موجود سيكون (*p) 4 بايت او في حال انظمة 64 بت ستكون المحاذاة علي 8 بايت وبالتالي شكل الذاكرة سيكون كالتالي

```
//for 32 bits system
 struct info{
     char *p;//4 bytes
     char ch;//1 byte
     char pad[3];//3 bytes
     int num;//4 bytes
 };
```

- هنا حجم الحشو يعتمد علي العنصر الذي يليه لنري المثال التالي

```
struct info(
     char *p;//4 bytes
     char ch;//1 byte
     short num;//2 bytes
 };
```

شكل الذاكرة سيكون

```
struct info(
     char *p;//4 bytes
     char ch;//1 byte
     char pad;//1 byte
     short num;//2 bytes
 };
```

- ما حدث انه بعد المتغير (char) كنا نحتاج الي حشو بمقدار 3 بايت ولكن وجد ان العنصر الذي يليه من نوع (short) يحتاج 2بايت لذا سيكون الحشو بقدار 1 بايت وبعده المتغير الاخر وبذلك حققنا شرط المحاذاة للـ short

-بمعني انه اذا امكننا تحقيق شروط المحاذاة التي ذكرت سابقا فهي الاهم من الحشو

- لنجرب البرنامج التالي

```c
/*
 *structure padding 1.c
 *
 * Created: 9/21/2016 8:02:52 PM
 *  Author: mohamed hussein
 */

#include <stdio.h>
#include <stdlib.h>

struct a{
    char ch1;
    short count;
    int num;
    char ch2;
};
int main()
{
    struct a x={'a',0xabcd,0x11223344,'b'};
    unsigned char *p;
    int i=0;
    p=&x.ch1;
    printf("the size of struct a is %d\n",sizeof(x));
    printf("the address of struct a is %x\n",&x);
    puts("content of struct memory block");
    puts("-------------------------------------------------");
        for(i=0;i<sizeof(x);i++)
        {
        printf("address [ %x ] contain  [ %x ] as hexadecimal and [ %c ] as
ASCII\n",p+i,*(p+i),*(p+i));
        }

    return 0;
}
```

- النتيجة

```
the size of struct a is 12
the address of struct a is 28ff0c
content of struct memory block
-------------------------------------------------
address [ 28ff0c ] contain  [ 61 ] as hexadecimal and [ a ] as ASCII
address [ 28ff0d ] contain  [ 19 ] as hexadecimal and [ ↓ ] as ASCII
address [ 28ff0e ] contain  [ cd ] as hexadecimal and [ ═ ] as ASCII
address [ 28ff0f ] contain  [ ab ] as hexadecimal and [ ½ ] as ASCII
address [ 28ff10 ] contain  [ 44 ] as hexadecimal and [ D ] as ASCII
address [ 28ff11 ] contain  [ 33 ] as hexadecimal and [ 3 ] as ASCII
address [ 28ff12 ] contain  [ 22 ] as hexadecimal and [ " ] as ASCII
address [ 28ff13 ] contain  [ 11 ] as hexadecimal and [ ◄ ] as ASCII
address [ 28ff14 ] contain  [ 62 ] as hexadecimal and [ b ] as ASCII
address [ 28ff15 ] contain  [ 4b ] as hexadecimal and [ K ] as ASCII
address [ 28ff16 ] contain  [ 4d ] as hexadecimal and [ M ] as ASCII
address [ 28ff17 ] contain  [ 0 ] as hexadecimal and [   ] as ASCII
```

-شكل الذاكرة وفقا للنتيجة

ملاحظات	المحتوي	العنوان
متغير char ch1	‘a’	0x28ff0c
حشو	padding	0x28ff0d
متغير short count	0xcd	0x28ff0e
	0xab	0x28ff0f
متغير int num	0x44	0x28fff0
	0x33	0x28fff1
	0x22	0x28fff2
	0x11	0x28fff3
متغير char ch2	‘b’	0x28fff4
حشو 3 بايت	padding	0x28fff5
	padding	0x28fff6
	padding	0x28fff7

- ماذا لو لدينا struct بها احد عناصرها عبارة ايضا عن struct هنا الـ structالداخلية ستقوم بعمل محاذاة علي حسب اكبر متغير سواء داخلها او خارجها ونفس الحال بالنسبة للـ struct الخارجية

```
struct test{
     char ch;//1 byte
     struct _inner{
         char c;
         int num;
     }inner;
 };
```

- سنجد هنا الـ struct الداخلية اجبرت الـstruct الخارجية علي المحاذاة علي 4 بايت

```
struct test{
     char ch;//1 byte
     char pad1[3];
     struct _inner{
         char c;
         char pad2[3];
         int num;
     }inner;
 };
```

- اعادة ترتيب الـ Structure

- اسهل طريقة لاعادة ترتيب الـ struct لتقليل الحشو هو ترتيب عناصرها بشكل تنازلي من حيث المساحة التخزينية

فمثلا

```
struct T{
    char c;
    struct T *p;
    short s;
};
```

ستاخذ الشكل التالي في حال نظام 64 بت

```
struct T{
    char c;//1 byte
    char pad[7];//7 bytes padding
    struct T *p;//8 byte
    short s;//2 byte
    char pad[6];//6 bytes padding
};
```

وبالتالي تكون المساحة الكلية 24 بايت, وفي حال اعادة ترتيب العناصر كما قلنا

```
struct T{
    struct T *p;// 8 bytes
    short s;//2 bytes
    char c;//1 byte
    char pad[5];//5 bytes
};
```

وبالتالي تكون المساحة الكلية 16 بايت فقط, قد تبدو تلك المساحة التي تم توفيرها صغيرة 8 بايت فقط ولكن لنفترض ان لدينا قائمة مرتبطة (linked list) بها اعداد كبيرة ستصبح الـ 8 بايت لها قيمة وقتها

*قد لايؤدي اعادة الترتيب الي توفير المساحة وهذا حسب شكل الـstruct

*يفضل مراعاة ان اعادة ترتيب تلك العناصر لا يؤثر بالسلب علي مدي وضوح كود الـ struct للقارئ او readability

*يجب مراعاة عدم استخدام متغيرات اكبر من الاحتياج

- الغاء شروط المحاذاة (alignment rules)من خلال Packing

-يتم ذلك من خلال استخدام بعض الـ preprocessor لمنع مترجم السي compiler من عمل padding

```
struct __attribute__((__packed__)) mystruct_A {
    char a;
    int b;
    char c;
};
```

*بعد تجربتي لهذا الكود وجدت انه يمنع الحشو في نهاية الـstruct فقط

المراجع :

stack overflow-

The Lost Art of C Structure Packing-